FTPMAN Design for Local Stations

Implementation considerations

Oct 24, 1991

The Fast Time Plot Manager has been implemented for the Local Stations as part of the Linac controls upgrade. It is a "local appli cation" to keep from placing the code into the system itself and to facilitate writing it in a high level language. For optimal integration with the local station system code, however, some special accommodations were added to the system. These notes describe the relevant considerations and the solution implemented.

FTPMAN is a data request protocol. The local stations already supported three data request protocols (Classic, D0 and RETDAT), so this is the fourth. For the new Linac, devices will be entered into the central database as having a common source node—the Server Node ($0601). Because fast time plots will be made of readings or settings of the same devices normally entered on a parameter page, FTPMAN protocol support must include server support, which is automatically provided when an FTPMAN request is received that includes devices from other nodes. The server node then forwards the request to the "real" target nodes (using group addressing if more than one other node is men tioned). These nodes recog nize that the request thus received need *not* be given server support because either all devices in the request are local, or the request was group addressed. The server node receives the replies and builds *its* reply in keeping with the order of the devices given in the original request. It replies to the requesting node according to the period specified in the original request.

For the RETDAT protocol, support consists of two parts. For non-server requests, the Accelerator Task receives network messages directed to RETDAT. A request is initialized and the first reply issued right away. Subsequent replies are issued at Update Task time early in the 15 Hz cycle. For server requests, the Accelerator Task receives the request directed to RETDAT, and it queues the request again to the network either to a single node or to a group address, depending on how many different non-local nodes are represented in the request message. First-time replies from the contributing nodes are sent right away. In turn, the server node's first reply is sent as soon as these are received. After the reply, during Server Task processing late in the cycle, all accumulated server replies which are due are sent to the requesting nodes.

In order for the Update Task to build non-server replies, or for the Server Task to send out its replies, a linked list of active requests is scanned. All non-server requests and all server requests are in the same linked list. The Update Task only reviews non-server requests, and the Server Task only checks for server requests. In order to support FTPMAN in a manner that integrates well into the system, such requests must get similar treatment.

linked list of active requests is the key to providing optimal assembly of network messages into common network frames. The linked list is main tained in an order which is sorted according to requesting node. Therefore, the answer messages generated during Update Task and Server Task pro cessing are queued in an order that is compatible with the network trans mit ting logic, which combines *consecutive* messages destined for the same desti nation node (and using the same 802.2 SAP#) which can fit in a maxi mum size frame.

As a local application, FTPMAN would not, until now, have easy access to such support. Each local application is called as a procedure by the Update Task early in the cycle. But to provide server support, it needs to be called at Server Task time as well in order to deliver server answers at that time. It cannot wait until the Update Task time in the next cycle, because at that time, non-server replies are being returned that may contribute to building server replies.

Generic request protocol support

From analysis of the data request support already completed for both the D0 and accelerator request protocols, there are only a few hooks into the rest of the system needed to provide support for any acnet-header-based request protocol.

The protocol package must be invoked by network messages that are received. This has been done until now by supplying a new *task* that waits for a network message using the `NetCheck` routine that is part of the network layer. As soon as the Acnet Task processes the message, it is placed into a message queue for that task, which was specified in the task's original call to `NetCnct`, on which the task waits via `NetCheck`. For local appli cations, which are *not* tasks, there is only a procedure call interface, so this method will not work. The Update Task calls each local appli cation during data access table processing once per 15 Hz cycle. The following is the new scheme for calling the local application in response to a received message from the network.

The Accelerator Task already waits on a message queue that receives all RETDAT and SETDAT requests, delivered by the Acnet Task. It uses a common message queue to insure that requests and settings are not processed out of sequence. The new scheme expands on this by permitting a local application to request that an additional protocol *also* be received into this same message queue. When the Accelerator Task finds a message which is destined for neither RETDAT nor SETDAT, it searches a new protocol table for a matching entry that contains the protocol's destination task name for requests, such as FTPMAN. From that matching entry, it obtains a pointer to a local application's LATBL entry, which is used to invoke the local application, indicating to it that a network message was received, allowing the local application access its own static variables and other parameters. In order to pass a reference to the received message, a pointer to the

the message, it returns to the Accelerator Task, which checks the queue for any more messages.

In addition to providing a connection so that a local application can be notified of network messages directed to it, hooks are also provided that allow it to fulfill requests during Update Task and Server Task processing.

When the Update Task scans the chain of active data requests, it calls a routine to process each one. That routine is given only a pointer to the request block used for that protocol. The Update Task keys on the request memory block type# in order to know what routine to call. It does this for two cases. For newly-received non-server requests, there is an attempt to make the first reply right away. The others will follow according to their request periods. For the accelerator protocol, for example, these routines are called ACUDPNEW and ACUPDCHK.

The Server Task, running late in the cycle, makes a similar scan of the active request chain to decide what routine to call for updating a server request. It also keys on the request block type#. It is necessary to distinguish whether the call to update is for the non-server case from Update or the server case from Server.

When the QMonitor Task determines that an acnet-header message has been completely transmitted, and the optional transmit status has been delivered to the user's variable, it calls a routine based upon the NETQFLG word in the message block. If bit#6 is set in the hi byte of that word, then the pointer to the parent request block, which is at a fixed location within the reply block message structure, is used to obtain the request block type# that, via a search of the protocol table, makes it possible to invoke the update procedure that is part of the local application but separately called. The arguments are a call type# and a pointer to the request block.

One more piece of generic information about a new protocol that is needed is a means of determining the destination node for a reply to a protocol's request. The INSCHAIN routine uses it when it inserts a request into the active request chain, which as noted before, is maintained in requesting node order.

The protocol table entry that is registered by a local application to satisfy the above hooks includes:

Request block type#
Offset to requesting node in request block
Ptr to handler which can be called by Update, Server, or QMonitor
Network task name (such as FTPMAN)
Ptr to LATBL entry that specifies parameters for calling LA.

```
   CallType= (delChk, updNew, updNServ, updServ);

   PROCEDURE Handler(call: CallType; VAR rBlk: ReqBlock);
```

Here `rBlk` is the allocated request block of the type# in the protocol table.

The set of routines for managing the protocol table, found in the OPENPRO module of the system code, are as follows:

```
   PROCEDURE InitPro;  {  Initialize protocol table at reset time }

   FUNCTION OpenPro(mBType: Integer; rNOff: Integer; Handler: ProcPtr;
                    taskName: Longint; LAEntry: ParamPtr): Integer;
                    {  Place new entry into protocol table }

   PROCEDURE ClosePro(mBType: Integer); { Remove entry from table}

   FUNCTION HandlerPro(mBType: Integer): ProcPtr; {  Get Handler
address}

   FUNCTION RNodePro(mBType: Integer); Integer; {  Get requesting node }

   FUNCTION LAEntPro(taskName: Longint); LAEntPtr; {  Get LATBL entry
ptr}
```

Details

The Update Task calls local application FTPM during Data Access Table processing. During the “init” call, static memory is allocated and initialized. FTPM gets the ACRQ message queue id via the `Attach_X` call to pSOS. It calls `NetCnct` to cause messages for FTPMAN to be directed to the same queue waited on by the Accelerator Task. FTPM also installs an entry in the protocol table, providing both a handler (for fulfilling requests and checking for cancelling one-shot requests) and the ptr to the parameters area in the LATBL entry. The handler accepts two arguments: a call type# and a ptr to the request block as found in the active request chain and as described above.

After initialization, the calls to FTPM from the Update Task every cycle may not be useful. However, in order to cancel one-shot requests, it may be convenient to mark the request block to be cancelled by the cycle call to the LA. This is because cancellation may require access to the static variables of the LA, and they are not accessible from the update procedure, although one could consider including a pointer to the static variables area for that purpose in the request block.

Upon termination, if the LA is disabled, the protocol table entry must be cleared using `ClosePro`, and `NetDcnt` must be called to stop queuing further messages into the ACRQ message queue. Finally, the static variables area must be released.

When a network message is received by the Accelerator Task via the ACRQ message queue, and it is neither RETDAT nor SETDAT, the protocol table is scanned for a match on FTPMAN. From the matching entry is obtained the ptr to the LATBL entry which is all that is needed to invoke FTPM. By placing into the parameters area a ptr to the message queue contents just read, FTPM can find the network message and process it in the network frame buffer.

The Update Task, when following the linked list of active requests, calls the handler found in the protocol table entry for the request block type# encoun tered. The handler updates the non-server request, and it should queue an answer message to the network if due. Later in the cycle, the Server Task also follows the active request chain and calls the handler to update and deliver any server-type requests that are due. When the QMonitor Task runs and finds that an Acnet header-based message has just been transmitted, it also calls the handler to check for automatic cancellation of a one-shot request.

In summary, then, the FTPM local application is called by the Update Task during data access table processing each cycle. It is called by the Accelerator Task to process a network message when it arrives. The handler whose entry point is registered in the protocol table is called by the Update, Server and QMonitor Tasks when appropriate to build replies and to delete one-shot requests.

Time-stamps

The first version of this FTPMAN support has a serious problem in that the time -stamps that are recorded with each data point are not correct. It is required of every front end that supports FTPMAN that the reference for the time-stamps be the clock event 02 of the Tevatron clock. This allows the fast time plot program to correlate data points collected from different front ends on the same plot. But the local stations do not yet have access to this clock event signal, which occurs every 5 seconds. There are two solutions to correct for this.

The first is to install hardware to decode this clock event signal that would make it available to a local station. If one local station had this information, it could use the network to send it to all the rest. It might do it only once per minute, rather than every 5 seconds. Local stations can count 15 Hz cycles as well as any node. What is missing is the proper phase. As a result, it might take up to one minute before a newly-reset local station was able to furnish correct time-stamps for fast time plots. A disadvantage of this approach, besides the need to build the hardware, is that it makes one node more important than the rest, which is not in the spirit of distributed systems.

The second solution is to design another FTPMAN protocol variant which would include in the request a suggested starting value for a time-stamp. This solution is a software-only solution. The FTPMAN requester must know about these clock events anyway, because it must interpret the time-stamps to plot the data. But the requester will not know if a front end supports the new variant. So the logic might proceed by trying the current continuous plot standard request format. If an “unsupported typecode” response is returned, it could switch to using the new variant. This might seem messy, but it is unlikely that other front ends that *do* have access to clock event 02 would implement support for the new variant.

The current version of FTPMAN for the local stations is about 1200 lines of Pascal source code that compiles into less than 6K bytes.